CONTROL STRUCTURES FOR HIGH SPEED PROCESSORS

by

Gary K. Maki
Robb Mankin
Patrick A. Owsley
Guihang Moon Kim

Electrical Engineering Department
University of Idaho
Moscow, Idaho 83843

NASA Grant
NAG 5-93

ABSTRACT

A special purpose processor was designed to function as a Reed Solomon decoder with a throughput data rate in the Mhz range. This data rate is significantly greater than is possible with conventional digital architectures. To achieve this rate, the processor design includes sequential, pipelined, distributed, and parallel processing.

The processor was designed using a high level language RTL (register transfer language). RTL can be used to describe how the different processes are implemented by the hardware. One problem of special interest was the development of dependent processes which are analogous to software subroutines. For greater flexibility, the RTL control structure was implemented in ROM.

The special purpose hardware required approximately 1000 SSI and MSI components. The data rate throughput is 2.5 megabits/second. This data rate is achieved through the use of pipelined and distributed processing. This data rate can be compared with 800 kilobits/second in a recently proposed VLSI design of a Reed Solomon ENCODER[1].

I. INTRODUCTION

A working design that implements the features of sequential, pipelined, distributed and parallel processing is described in this paper. This processor consists of seven unique modules that operate asynchronously. Each module displays the characteristics of sequential, pipelined, distributed, and/or parallel processing. The state control within each module specifies the desired mode of operation. A major part of this paper is to describe control mechanisms that were used to implement the various modes of operation.

The processor function is to decode Reed Solomon Codes over GF(2**8). Each code word consists of up to 255 8-bit symbols and can correct up to 16 symbol errors. The Reed Solomon code is known for its powerful error correcting capabilities and has gained much recent attention. A recent VLSI design of a Reed Solomon encoder details some of the applications[1]. The reader can refer to a coding theory textbook such as Peterson and Weldon[2] for details of cyclic codes. It is not necessary to understand coding theory nuances to appreciate the results presented in this paper.

Following is a definition of the processing requirements of the modules in general terms.

i) Simple serial to parallel conversion of the input data stream. The 8-bit symbols are stored in buffered RAM.

ii) Calculate 32 syndrome vectors by solving 32 equations of order 254.

iii) Formulate a 16 by 16 matrix and determine the rank t, with t less than or equal to 16.

iv) Solve t simultaneous equations.

v) Evaluate 255 equations of order t.

vi) Evaluate t equations which is the division of two polynomials of order t.

vii) Correct output data and present correct results.

All of the above operations must be performed in the Galois Field GF(2**8). The operations in GF(2**8) are 8-bit modulo 2 addition and multiplication in the field of polynomials modulo f(x) = X**8 + X**4 + X**3 + X**2 + 1. The addition operation is easily implemented. However, the multiplication operation must be accomplished through the use of logarithm and anti-logarithm tables. These tables result from the fact that the code is cyclic. Multiplication is accomplished with these tables using modulo 255 addition.

II. DESIGN APPROACH

The completed system required about 1000 SSI and MSI components. Naturally when a design of this magnitude is undertaken, it is impossible for the designer to formulate the final implementation using low level logic design tools such as logic diagrams. It is necessary to use a high level language to properly focus attention on the design problems and avoid the unnecessary distractions of specific hardware details of realizing individual chips. The language used in this design is one developed at the University of Idaho but is not unlike many other design languages that are in existence[3]. An important feature of this language is the ability to allow the designer to remain conscious of the control structure of the machine. Access to the control structure is important

in order to specify the mode of processing and to distinguish between sequential, parallel, pipelined, and distributed processing. This design langauge has relatively simple constructs to allow the designer close association with the control structure. Basically, an RTL statement has the following structure.

<control expression>: <list of actions>

<Control expression> is a boolean expression which can be easily implemented using any of several standard control structures. <List of actions> is a set of unconditional transfers, register transfers, conditional transfers, and control modification statements. Evaluation of the statement proceeds as follows: whenever the control expression is evaluated TRUE (i.e., <control expression> = 1) all transfers within <list of actions> become active, otherwise no transfers will occur.

The above basic statement can be modified by use of the IF-THEN-ELSE conditional statement. This modification allows for more flexibility for the designer without sacrificing control consciousness. The basic form of the IF-THEN-ELSE construct is as follows:

```
<control expression>: <list of actions> (1);
                      IF <rel expression> THEN <list of actions> (2);
                                               <list of actions> (3);
```

In essence the procedure to implement the above structure would be as follows:

```
<control expression>: <list of actions> (1);
<control expression>*<rel expression>: <list of actions> (2);
<control expression>*<rel expression>': <list of actions> (3);
```

An example of the control structure in the RTL is shown next:

```
S5*CK:  SPTR - 1 -> SPTR              /* Decrement SPTR        */
        0 -> S5                       /* <list of actions> (1)*/
        IF SPTR = 0 THEN 1 -> S6      /* <list of actions> (2)*/
                    ELSE 1 -> S1      /* <list of actions> (3)*/
```

This statement would be evaluated as follows:

```
                S5*CK:  SPTR - 1 -> SPTR
                        0 -> S5
(S5*CK)*(SPTR=0): 1 -> S6
(S5*CK)*(SPTR≠0): 1 -> S1
```

The hardware to implement the control structure utilizes ROMs, as depicted in Figure 1.

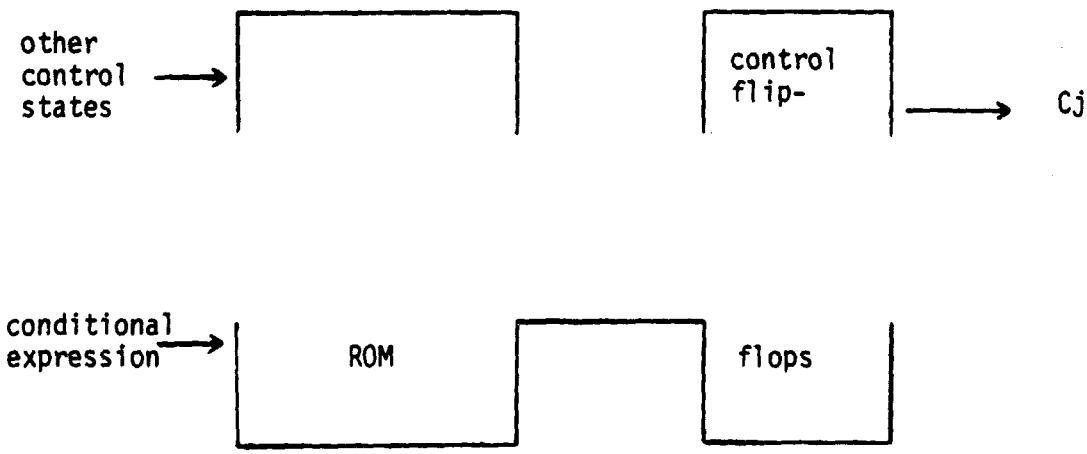

Figure 1. Hardware Implementation of Control

Following is an example of RTL implemented with a ROM controller.

```
DECLARE (A,B,D) Register, (COUNT, C) Counter
     C0:  IF <GO> = 1 THEN 1 -> C1, 0 -> C0
     C1:  A + B -> C, 0 -> C1, 1 -> C2
          IF <C=0> THEN COUNT - 1 -> COUNT
     C2:  C .AND. D -> C, 0 -> C2, 1 -> C3
     C3:  C + 1 -> C, 0 -> C3, IF <COUNT = 0> THEN 1 -> C4
                                              ELSE 1 -> C1
     C4:  IF <READY> = 1 THEN 1 -> C0, 0 -> C4
                         ELSE 1 -> C4
```

The ROM to control this small process would consist of 8 inputs and 6 outputs. The inputs would be the control states {Ci}, i = 0,1,...,4, and the signals GO, COUNT = 0, and READY. The outputs would drive the control state flip-flops and conditional decrement of COUNT in control state C1. All unconditional transfers would be enabled by the control state flip-flops.

The chief advantages associated with using this structure include reduced hardware and flexibility. During the design process it is not uncommon to discover design oversights or to require a modification in the design algorithm. With the control programmed into a ROM, these modifications can be more easily implemented. Another desirable feature, which will become more apparent later in the paper, is that one can change a sequential process into a pipelined process through reprogramming the control ROM. This assumes that the necessary holding registers are available to allow for pipelined data flow. In the processor

described here, ROMs were used. PLAs or PALs, with internal flip-flops, could be used and would have served to implement the control more efficiently.

Definition: A process P is a set of operations that is specified by a set of control states {Ci} which define a sequence of register operations.

A process can assume any of the following modes: sequential, pipelined, parallel, or distributed. The control states specify the mode desired. Following is the specification of these modes of operation.

In the sequential mode, the control structure is

Ci: Ci inactive, Ci + 1 active.

In this mode of operation, successive control states are normally assumed. Furthermore, only one control state is active at any one moment. The RTL example above illustrates the sequential process.

In the pipelined mode, the general control structure is

INITIAL STATE C0: IF <start$expression> = TRUE THEN C1 active.
INTERNAL STATES OF THE PROCESS Ci: Ci -> Ci + 1
END STATE: IF <end$expression> = TRUE THEN C1 inactive

The pipelined process is initiated whenever the start expression is true and then state C1, the first state of the pipelined process, is activated. Once C1 is active, then successive stages of the pipelined process become active. The pipelined process is inactivated when the end expression becomes true and then C1 is made inactive, which in turn inactivates the successive stages of the process. As distinguished from the sequential process, many control states are active at the same instant of time.

An example of a pipelined process is given below.

```
DECLARE (A,B,C,D,E,F) REGISTER, COUNT COUNTER
C0:  IF <GO = 1> THEN 1 -> C1, 0 -> C0
C1:  A + B -> C, C1 -> C2
     IF <C = 0> THEN COUNT - 1 -> COUNT
     IF <COUNT = 0> THEN 0 -> C1
                    ELSE 1 -> C1
C2:  C2 -> C3,  C .AND. D -> E
C3:  E + 1 -> F. IF <C2 = 0 AND C3 = 1> THEN 1 -> C4
                                        ELSE 0 -> C4
C4:  IF <READY = 1> THEN 1 -> C0, 0 -> C4
                    ELSE 1 -> C4
```

State C0 is the initial state and C1 the first state in the pipeline. C1 also serves as the end state in that information concerning when the process is to terminate is determined in C1. Control hardware for this process is implemented with a ROM or PLA. Note also that this pipelined process is functionally equivalent to the sequential process listed above. The differences are associated with the control and the extra registers to allow pipelined data flow.

For parallel processing, consider the control set of states {Ri} and {Si}, where R0 and S0 are the initial states of parallel processes R and S. Both processes are initiated as follows:

```
R0:  IF <begin$expression> = TRUE then R1 active, R0 inactive.
S0:  IF <begin$expression> = TRUE then S1 active, S0 inactive.
```

Each process can be initiated asynchronously and both processes can be active. Each process can be sequential or pipelined. For example, both of the RTL examples above could be activated to operate in parallel.

One of the challenges in hardware design is to implement a process similar to a subroutine in software. Several processes of this nature, which could be termed dependent processes, were implemented in the design presented in the paper. The problem of initiating a dependent process is not difficult for it would involve only making the <begin$expression> evaluate true. The challenge comes in providing a "return address."

Definition: A main process is one that is not called or initiated by some other process. A dependent process is one that is initiated by another process and returns control back to the process that does the initiating.

A dependent process can be initiated by several main processes or by one main process from several of its control states. The main process, after initiating a dependent process, can continue executing, or can suspend activity until the dependent process completes execution.

Definition: The control state in the main process which is to become active after a dependent process has completed processing is called the return control state.

One big challenge with designing a dependent process is to provide a mechanism to allow for the return control state in the main process to be activated. There are several possibilities. First is to implement that which is done in computer software by providing a RAM that will store the proper return control state. This is most general and allows

the greatest flexibility but at the expense of hardware. Normally the degree of flexibility that this approach allows is not required in special purpose hardware implementations since the return control states are relatively few and well defined.

The approach used by the authors is that of setting one of several state flip-flops available to the dependent process that would specify the return control state in the main process. The disadvantages with this approach is reduced flexibility and hardware defined return control states. Since the number of return control states is small, the hardware benefits outweighed the general approach.

If main process activity is to be suspended, then a simple approach to the design of the dependent process is to provide no return control state. The main process simply would enter a control state that would wait until the dependent process is complete. An example of this type of control is

Ci: IF <Dependent$Process$Complete> = TRUE THEN Cj active, Ci inactive
ELSE Ci remains active

This approach is useful for those applications where a dependent process is initiated from only one main process and the number of return control states in the main process is relatively large. On the surface it would appear that the major cost is mutual exclusion of processing between the main and dependent process.

In considering this in more detail, let mutual exclusion of processing meet either of the following conditions: Let M and D denote the Main and Dependent processes respectively.

a) Time mutual exclusion where the hardware elements (registers, memories, etc.) that M and D both have access to are not being used by M and D at the same time.

b) Hardware mutual exclusion where the hardware elements of M and D are accessible to only one process.

If M is suspended then time mutual exclusion is insured and indeed only one process is active at any one moment. If hardware mutual exclusion is true, then both the main and dependent processors can operate in parallel. M can initiate D and then continue to process until it is ready to utilize the results of D, at which time it could check the status of D to determine if it has completed the process. It is possible to combine both hardware and time mutual exclusion. M and D can share hardware and therefore both cannot attempt to use that hardware at the same time. M in general has hardware that is not available to D. Therefore, M can initiate D and then process until a control state is entered that would require the use of hardware that D utilizes. Upon entering that control state, M must wait until D is complete and operate in the time mutual exclusion mode, where prior to entering this control state M operated in the hardware mutual exclusion mode. The processor designed here has operated in all three modes: time mutual exclusion, hardware mutual exclusion, and combined time and hardware mutual exclusion.

III. FAULT DETECTION

An 8085 microprocessor-based system is provided in the system to provide for input/output operations between the user and the system and

to act as an interface for running diagnostic tests. The operating system of the microprocessor has a built-in set of tests that can be invoked. The operator specifies the test data that will be used, the module in which to insert the test data, and the module from which the data is to be observed. For a built-in test set, a known output response is expected. If the desired output does not occur, then an error signal is given along with diagnostic information that can be useful for determining the location of the fault. The operator also has the option of specifying the input test set. If the system is operating in this mode. then the observed output is presented on a CRT screen. This feature allows for powerful diagnostic tools to be available to the user, where test data can be inserted at any point in the processor and the results observed at another point.

REFERENCES

1. K. Y. Liu, "Architecture for VLSI Design of Reed Solomon Encoders," pp. 170-175, IEEETC February 1982.

2. W. W. Peterson and E. J. Weldon, ERROR CORRECTING CODES, MIT Press, 1972.

3. S. G. Shiva, "Computer Hardware Description Language," pp 1605-1615, Proceedings of IEEE, December 1979.